

Trimester 1 2020

SENG 3011

DESIGN

DETAILS

CODE-ONAVIRUS



By: Kenvin Yu, Ezra Eyar, Victor Liu, Evan Lee and Eric Tan

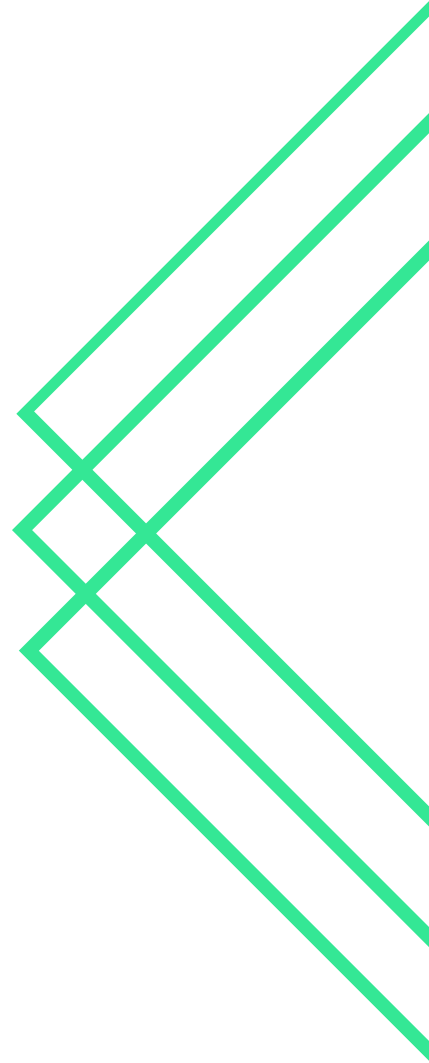
About Us

In light of recent events, the UNSW Medical Science faculty has reached out to our team at Code-onavirus to develop a RESTful API web service in order to track disease outbreaks around the world. The aim of developing such technology is to enable universal access to up-to-date information about outbreaks in order to control and prevent epidemics in future.

The following report details and describes the planned design of the API to be implemented to holistically explain the framework of the system. Such framework exploration include rationale for language, hosting and storage decisions as well as examples of API usage.

Our Vision

Our final aim is to generate an API that is accessible and easily usable for any individual aiming to develop web applications. Moreover we aim to also implement a web application using our developed API to interactively inform and warn individuals of global disease outbreaks.



Development Plan

Our software solution will be developed according to agile methodologies and will adhere to a Model, View and Controller (MVC) architecture in order to deliver a RESTful API web service. Our solution contains two main subsystems: a web scraping component to gather articles and disease reports from our main data source, and a web server to facilitate our RESTful web service. Our system will be deployed in the DigitalOcean cloud environment to ensure wide availability with minimal delay and consistent reliability. We will be primarily developing our software solution in Python. For a justification of technologies mentioned, refer to the relevant section below.

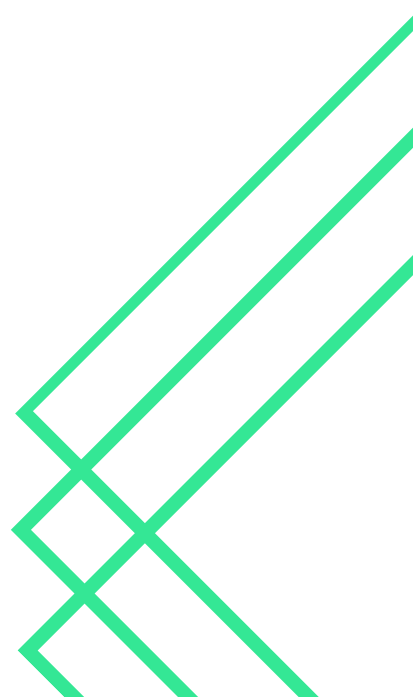
Our web scraping subsystem delivers the 'model' component of the MVC architecture. The primary objective of this subsystem is to gather articles from our main data source, the World Health Organisation's (WHO) disease outbreak news, and to then interpret and extract useful metadata and disease reports. We will be using Scrapy as our web scraping library to perform the collection of article data, as well as the Google Places API to assist in classifying location data. The subsystem will then store these processed articles and disease reports in a central database, which enables the the web server to independently query and fetch relevant data from this database without requiring the web scraper to operate on each API call. This reduces the response time of our API module and offers increased reliability by eliminating the single point of failure in the event that WHO's website is unavailable. As a result of caching, the web scraping module needs only to be executed daily or bidaily to populate the database with new articles

We also intend to supplement our API by collecting data from the public Johns Hopkins University Center for Systems Science and Engineering (JHU CCSE) COVID-19 database, which has no articles but includes daily reports in a csv file format that does not require web scraping. This will supplement our database by providing more specific information on COVID-19 that would have been missed by the aforementioned WHO reports. These reports will also be parsed and stored in the central database.

Running In Web Service Mode

Our system will be utilising MongoDB as the aforementioned NoSQL database management system. In particular, our software solution relies on MongoDB Atlas which is a hosted cloud database service that employs MongoDB. This choice aligns with our prioritisation of reliability in our API module over a minor loss in speed.

The 'controller' component in our system will be performed by a Flask web server. Flask is a lightweight Python web framework and in conjunction with its extension, Flask-RESTPlus, will be suitable for facilitating our RESTful web service. Our users' API calls will be received by the Flask server which will then query our MongoDB database for the relevant article data to return back to our users through a JavaScript Object Notation (JSON) format. For deployment in production, we will be running our Flask application using uWSGI with NGINX as a reverse proxy for more robust long-term use. The 'view' component in our MVC architecture is provided by the Flask-RESTPlus extension which automatically generates API documentation using Swagger UI.



Passing Parameters

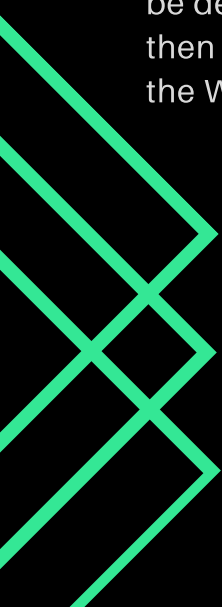
Web Scraping

In terms of scraping the web, WHO articles have a relatively consistent structure in that the location of the disease outbreak is in the title. Similarly, the date of the outbreak is always consistently after the title and hence makes it easier to identify when scraping the article. Hence all the parameters required in the payload of the API will be relatively easy to extract and store in our database. In general, the most difficult JSON component to return will be the categorisation of a report based on symptoms.

In order to have the information readily available for the API, the web scraper will collect key data such as the name of the country and date from the article on the WHO article. These will then be stored inside the database alongside the article and associated reports. By collating everything together in the database, reports from the article will all be connected together to more easily return results to the user.

In relation to the supplementary JHU CCSE COVID-19 database, there will not be a need to scrape the database given there are no articles or reports. However, there are daily reports in a CSV format which will again be relatively easy to add to our MongoDB database given the consistency of the format of the CSV files. In order to collect the CSV files, we will be using Github to perform regular pulls of the repository in which the data is stored. These pulls are scheduled using Cron Job, a Linux utility that schedules tasks on a server. The regular pulls will ensure the data is consistently up to date.

In order to store the information from the supplementary database, an adapter will be developed in order to extract the data stored in the CSV files. This data will then be stored in the database to supplement the pre-existing data extracted from the WHO.



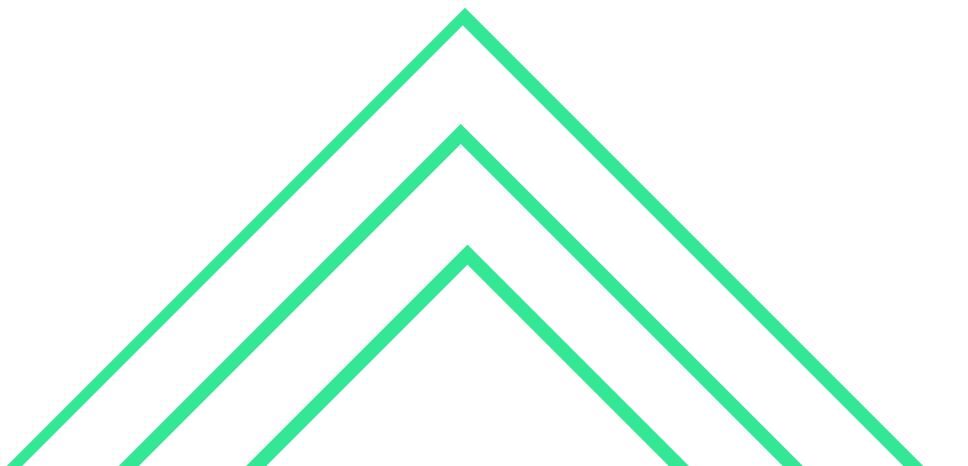
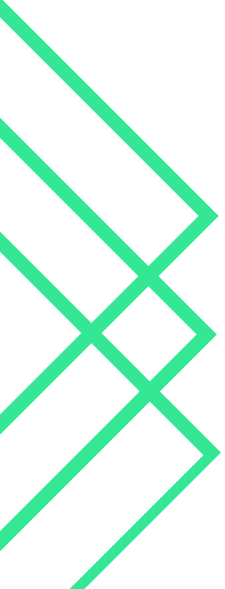
Passing Parameters

API Endpoint

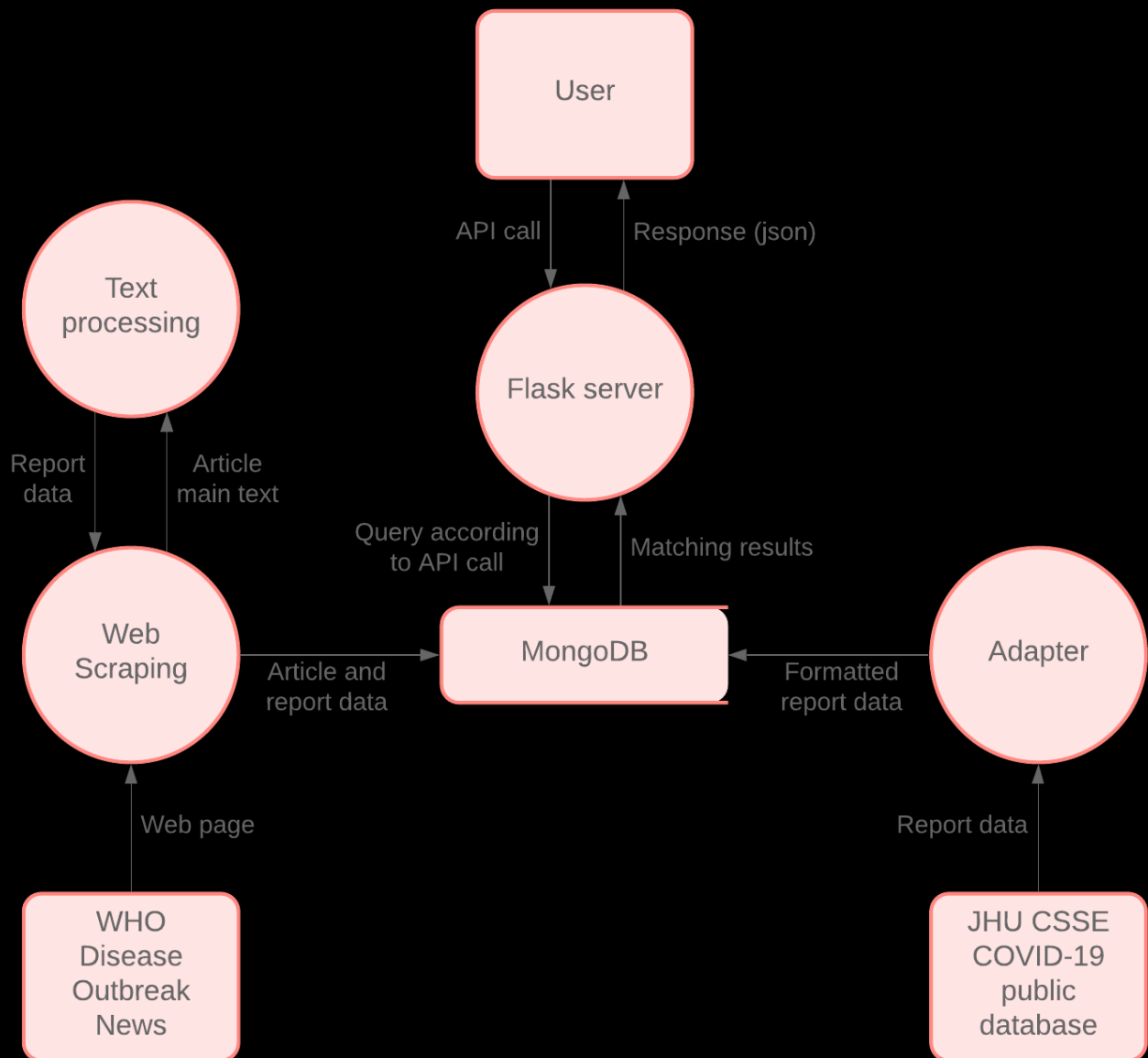
For the API endpoint, we will be utilising a payload within the body of the request as a PUT request. The rationale behind this decision is due to the fact that the PUT request is more universal given that the user can query multiple parameters at once resulting in more specific filtering of results, specifically meeting the needs of the user. Moreover, the GET request would be too convoluted, particularly if the user wants to query many parameters at once. The reason for this is that the GET query works through having parameters queried at the end of the address as opposed to having a payload in the body. Hence querying through more parameters would just be adding to the end of the query resulting in a very long and inefficient address endpoint for the user to access.

The result for the API call request will be presented to the user in a JSON format. The rationale behind using a JSON format is that most web-based applications implement JavaScript (JS) based frameworks such as React. Hence, making the API result more universally compatible widens the market of potential users, accomplishing our goal of having as many users as possible.

The results of the API will come directly from the database, depending on the parameters provided by the user in the payload of the API request. The API request will be processed and the search parameters required by the user will be queried to the MongoDB database. Then the result will be delivered to the user in the aforementioned JSON format, specifically a list of articles with each article having a list of reports connected to it. The user can then utilise this JSON return for their own use. An example of a query result is shown below in the success code section. Error codes and their meanings are entailed in the section following the success codes section.



Dataflow Diagram



API Examples

Sample WHO PUT request

```
curl -X PUT "https://codeonavirus.com/who/articles"  
-H "accept: application/json"  
-H "Content-Type: application/json"  
-d "{ \"start_date\": \"2005-08-26\", \"end_date\": \"2005-  
09-23\", \"key_terms\": \"Yellow fever, Dengue\",  
\"location\": \"Africa,,,\"}"
```

Sample JHU PUT request

```
curl -X PUT "https://codeonavirus.com/jhu/daily_reports"  
-H "accept: application/json"  
-H "Content-Type: application/json"  
-d "{ \"start_date\": \"2020-02-20\", \"end_date\": \"2020-  
03-07\", \"state\": \"Hubei\", \"country\": \"China\"}"
```


API Request Codes

WHO Success Code

200 - The query to the endpoint has successfully been executed and a JSON is returned

The JSON is of the following format:

```
[
  {
    "url": "https://www.who.int/csr/don/2005_09_22/en/",
    "date_of_publication": "2005-09-22 xx:xx:xx",
    "headline": "Yellow fever in Burkina Faso and Côte d'Ivoire",
    "main_text": "WHO has received reports of an outbreak of yellow fever in Batie, Gaoua and Banfora districts in Burkina Faso in the southeast of the country, near the border with Côte d'Ivoire. Four cases including 1 death have been laboratory confirmed by Centre Muraz, Burkina Faso and by the WHO Collaborating Centre for Yellow Fever, the Institut Pasteur, Dakar, Senegal. The fatal case, a boy of 4 years old, came from Bouna region in Côte d'Ivoire. A team from the Ministry of Health and WHO in Burkina Faso and a team from the Ministry of Health, WHO and UNICEF in Côte d'Ivoire quickly investigated the outbreak in this cross border area characterized by increased population movements. A mass vaccination campaign is being prepared in both countries to protect the population at risk and to prevent the spread of the disease to densely populated urban settings. The WHO Regional Office for Africa is working with both Ministries to determine the most appropriate strategies for disease control in the cross border area and to raise funds for outbreak response activities.",
    "reports": [
      {
        "diseases": [
          "Yellow fever"
        ],
        "syndromes": [
          "Fever of unknown Origin"
        ],
        "event_date": "2005-09-22 xx:xx:xx",
        "locations": [
          {
            "continent": "Africa",
            "country": "Burkina Faso",
            "state": "Noumbiel Province",
            "city": "Batie"
          }
        ]
      }
    ]
  }
]
```

API Request Codes

JHU Success Code

200 - The query to the endpoint has successfully been executed and a JSON is returned
The JSON is of the following format:

```
[
  {
    "date": "2020-02-20",
    "entries": [
      {
        "state": "Hubei",
        "country": "Mainland China",
        "last_update": "2020-02-20T23:43:02",
        "confirmed": "62442",
        "deaths": "2144",
        "recovered": "11788",
        "latitude": "30.9756",
        "longitude": "112.2707"
      }
    ]
  }
]
```



API Request Codes

Error Codes

- 400 - Malformed request: This means that the endpoint and method are both correct in the API request but the body of the request does not follow the framework given resulting in errors.
- 404 - Incorrect endpoint: The web address called in the API request is incorrect resulting in an error. For example, setting the address as codevirus.com instead of codeonavirus.com.
- 405 - Method not allowed: The web address called in the API request is correct but the method being called does not exist on the endpoint resulting in an error. For example, calling a GET request when only a PUT request is possible from the endpoint.
- 500 - Internal server error: There is an issue with the server causing it to be down and unaccessible. For example, the cloud database goes down due to unforeseen circumstances resulting in an error.

Justification

Hosting - Digital Ocean

Our team will opt to host our web service on DigitalOcean to minimise backend concerns, particularly the maintenance and repair of servers. We will choose DigitalOcean due to their impressive average uptime of 99.99%, which is higher than the industry average for cloud hosting services at 99.94%, demonstrating the clear reliability of DigitalOcean's servers. Furthermore, DigitalOcean offers quick load times as a result of their Solid State Drive (SSD) based virtual machines. Their developer-friendly ecosystem offers highly customisable deployment environment options, and importantly allows our project to be easily scaled and upgraded if our service requires it. As with most hosting services, DigitalOcean employs daily backups to minimise the potential loss of data in the event of unfortunate catastrophes.

Operating System - Ubuntu

Our web service will operate on an Ubuntu server hosted by DigitalOcean. Unlike a Windows based server, Ubuntu is a free and open-source Linux distribution that provides a rich set of package groups as well as great support and documentation. Specifically, our web service will be deployed in Ubuntu 18.04 (LTS) to ensure stability and security over the long term.



Justification


Programming Language - Python

We will be developing our web service primarily in Python 3.6 as it is an easy to learn, versatile, general purpose programming language suitable for an agile development cycle. Python offers an extensive collection of libraries and third-party modules, including web frameworks and testing instruments, which our system will be employing to produce a web service. An important aspect of the API web service is the ability to create and handle JSON responses, which Python's in-built json module is perfect for.

Database - MongoDB

Our system will utilise MongoDB for storing and transmitting data in the JSON schema as required by the project specifications. As a NoSQL database, MongoDB delivers superior performance and scalability compared to traditional SQL databases. It operates on a dynamic schema designed to address rapidly changing structured data that arises from agile software development. We will also be using MongoDB Atlas, a global cloud database service that deploys a fully managed MongoDB to ensure greater reliability and data security.

We will be using the Python module, PyMongo, to facilitate communication between our Python system and MongoDB. PyMongo is the native Python driver for interacting with MongoDB and is specifically maintained by MongoDB engineers.



Justification

Libraries, APIs and Third Party Modules

Our system will utilise the Python third-party module, Scrapy, for web scraping. Scrapy offers a faster alternative than other web scraping libraries. It will simplify and streamline our workflow, which ensures that our development team's attention is focused on interpreting the gathered article data instead of web scraping itself.

To assist in processing and interpreting articles after scraping, we will be employing the third-party natural language toolkit (NLTK) module. The NLTK offers a suite of text processing libraries for classification, tokenization, tagging and parsing that will help us deliver a core functionality of our service by simplifying the understanding and extraction of disease reports from articles.

We will be using the Places API through the Python client for Google Maps Services to supplement the geolocation data extracted from articles. This will benefit our API by providing the ability to understand geolocational hierarchies and context to provide our users with as much relevant data as possible.

To ensure our software solution delivers valid and correct data to our clients, we will be using the pytest framework to extensively verify our collected data and API responses. This includes both white box testing such as ensuring that our web scraping components collect all the necessary information, and black box testing to confirm that our API web service's response is formatted according to the specifications outlined in our API documentation.

Justification

Libraries, APIs and Third Party Modules

Our RESTful API will be served using the micro and lightweight web framework, Flask, in conjunction with the Flask extension, Flask-RESTPlus. For Python, the main alternative web framework is Django, which is a full-stack web framework that is excessive for our needs. Also, Flask is a more suitable option due to its flexibility and support for extensions that accelerates the agile development of simple web services and applications. We will be using the Flask-RESTPlus extension as it adds support for quickly building RESTful APIs, as well as offering API documentation using Swagger UI. This will simplify and ease the process of building our web service.

For deployment, we will be using uWSGI to serve the Flask application as Flask is not suitable for a production environment standalone. Furthermore, we will be operating a NGINX server as a reverse proxy to the uWSGI application server, providing a more robust and secure solution for long-term usage. This opens up scalability of our web service as well as removing the load of SSL encryption off of the uWSGI server.



Justification

Libraries, APIs and Third Party Modules

Python provides an in-built logging module which we will use extensively to maintain a record of data input, processes and data output throughout our software solution. Logging is invaluable as it simplifies the process of debugging and diagnosis in the event of failure, which is especially common in the development phase. By keeping a log, our team can also optimise processes and components that are causing bottlenecks so that we can deliver the highest quality product to our clients.

The logging is applied to both the API endpoint as well as the web scraper. The reason for this is to ensure that all key components are working properly and if there are any errors, they can be easily identified and fixed preventing major outages of the system.

The logger naturally logs any errors within the system based on a hierarchy, which is set by the user. In this case it is set to info, meaning it will log all messages from information about the system all the way down to critical errors to ensure detailed information while staying concise.




Justification

Libraries, APIs and Third Party Modules

In terms of finding out place specifically, we are using the Pycountry library in conjunction with the Pycountry_convert extension for Pycountry. This will be used to ensure that the spelling of locations are correct and stored correctly in the database after they are parsed through the aforementioned Google places API. This is another easy built-in library to use and suits the requirements of the task making it the most efficient option for completing the task.

To help with the aforementioned Scrapy for Python, we will also be implementing BeautifulSoup, which is a Python library designed to pull data from HTML and XML files. Effectively, this makes scraping easier as Scrapy is limited in its ability to extract data from crawling the web, being more efficient at scraping the web and returning what it finds rather than extracting key information such as the title of the article which we require. Hence, combining both libraries together ensures the most efficient web scraping possible for the system.

The dates of the articles, although apparently in a consistent format at first glance, are actually extremely varied and hence we need a method with which to extract dates to store the articles with. This is where the DateParser package comes in. This package is designed specifically for finding and parsing dates from HTML pages, streamlining the process of parsing in dates regardless of the format provided on the web page. Hence, this further increases the efficiency of our scraping and parsing process.



Justification

Libraries, APIs and Third Party Modules

One key module that we will use for implementation is the regular expression, or re module, of python. The purpose of regular expression is to match text that we want to find with text that exists in the document. This allows us to find particular sentences or phrases that are similar without having to provide multiple cases by hard coding. The use of this module will be for to find all words with capital letters in the report, allowing us to easily parse these into the Google places API, given that places can only start with a capital letter.

The final package that we will be using is the Unidecode package for Python. Problematically, many places in the world, particularly those in Europe, have characters with accents or similar punctuation on them which are not recognised as English characters. Hence through Unidecode, they are decoded into regular English letters allowing them to be detected and processed by the Google places API. Without doing this, the Google places API would not be able to process certain places given that it does not recognise non-English characters.



Shortcomings

High Coupling Of The System

One of the main shortcomings with out API endpoint system is the reliance of each component on other components within the system, also known as high coupling. For example, the information in the database is reliant on the web scraper to provide accurate information to be updated daily. Similarly, our endpoint querying is dependent on the information in the database to return in result in the query. Hence, the system is highly coupled as any error in scraping or outage of the database could bring down the endpoint.

However, there is no easily solution to this problem. This is evident given that high coupling does not follow good software design principles and therefore should be avoided wherever possible. The reason why this issue is difficult to address is that scraping the articles daily would be inefficient and time consuming resulting in a less efficient experience for the user due to the increased time complexity.

Moreover, the data is formatted better and guaranteed to be more accurate as it is processed and stored in a database whereas if the document is scraped every time the API is called, then the information may not be guaranteed to be correct or formatted correctly given the variance between articles. This also ensures that the key terms will match with the terms queried. For example, COVID-19 is referred to as novel corona virus or nCov in earlier reports which could cause not all articles being returned to the user.



Shortcomings


Reports Per Article

Another main shortcoming of the system was the fact that it can only return one report per article. However, the report is still stored in an array and in future there is potential for expansion of the system to store multiple reports in the array. The reason why it was only possible to return one report was because of the complexity and time associated with collating individual reports.

For example, on a report about COVID-19, there would be a report for someone from Melbourne and another one for someone from Sydney. Having to process and individually store each report would be far too costly in terms of time given that many articles have numerous reports, remembering that there are over 2000 articles means that collating the database would take far too long.

Similarly, this applies to searching as well. The time complexity skyrockets from being $O(n)$ to $O(n^2)$ for a search by location given that searching for location of articles would only involve a pass through but to find individual reports would mean it would first have to find the relevant articles and then find the specific reports of a certain location.

However, the trade off is that the system will provide less specific information to users and since only the first article is taken, then key information may be missed. Hence, it would most likely be better, given the nature of the system, to implement multiple reports for each article in future if possible.



Challenges

Place Identification


One of the main challenges in developing the endpoint was the way in which we would extract the location of the report. Generally, the articles follow a relatively consistent structure but for some of the older reports, the structure greatly differs which makes extracting the location inconsistent if going only off of the format and taking out a certain sentence.

The way we addressed that was by extracting all text that started with a capital letter, as locations are all proper nouns and hence start with capital letters. These words in capital letters are then checked through the Google places API to ensure they are valid geographic locations. If they are, then they are stored in the database but are ignored otherwise. This also enables us to easily find the province or state of a location if they are not provided in the article.

Key terms

Another major challenge encountered was the location of key terms in the article. If we were to scrape each time, then finding key terms would further increase the time complexity of the operations leading to potentially frustrating user experiences if the user was to search for multiple key terms at once. Hence, we decided to make further use of our database to address this issue.

First, we set up a list of key terms to be searched for given that there was much ambiguity of what key terms were so by restricting it, it made it clearer what we were searching for. Then through scraping, we found the key terms from the list and then attached them to the article object that would be saved in the database. In doing this, search queries could be executed much quicker and without ambiguity in terms of which terms could be searched for.



Challenges

Date Parsing

As mentioned earlier, date parsing turned out to be much more challenging than expected. The reason for this is because the format of the dates were different for many of the reports. The variation was not dependent on any factor at all, but was merely just different across certain reports. Hence, there was no easy way to match up the location of the date in the article with any factor. Hence, date parsing turned out to be a major challenge.

The solution to the problem was to replace irrelevant information from the article and then reformat it in a way such that the information could be more easily interpreted. For example, if the date had 16th and 18th of October, it would be processed and changed to 16th October and 18th October so that it could then be processed by the Dateparser Python package.

Scraping

Scraping proved to be challenging as well given that, similar to the challenge of date parsing, the format of the reports provided by WHO were actually relatively inconsistent. For example, some reports had the main heading enclosed in `<h2>` tags whereas others would implement `<h3>` or even `<h5>` tags. Hence, there wasn't a set location and tags within which key information could be clearly found in.

In order to address this issue, we implemented the aforementioned BeautifulSoup module for Python which would essentially restructure the website into a format that would be easy to navigate. This removes the complexity associated with finding information and allows key data to be easily extracted.